



# Chasing the Weakest Failure Detector for k-Set Agreement in Message-passing Systems

Achour Mostefaoui, Michel Raynal, Julien Stainer

## ► To cite this version:

Achour Mostefaoui, Michel Raynal, Julien Stainer. Chasing the Weakest Failure Detector for k-Set Agreement in Message-passing Systems. [Research Report] PI-1981, 2011, pp.15. inria-00606918

**HAL Id: inria-00606918**

**<https://inria.hal.science/inria-00606918>**

Submitted on 7 Jul 2011

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

## Chasing the Weakest Failure Detector for $k$ -Set Agreement in Message-passing Systems

Achour Mostéfaoui<sup>\*</sup>, Michel Raynal<sup>\*\*</sup>, Julien Stainer<sup>\*\*\*</sup>  
{achour|raynal|jstainer}@irisa.fr

**Abstract:** This paper continues our quest for the weakest failure detector which allows the  $k$ -set agreement problem to be solved in asynchronous message-passing systems prone to any number of process failures. It has two main contributions which (we hope) will be instrumental to complete this quest.

The first contribution is a new failure detector (denoted  $\Pi\Sigma_{x,y}$ ). This failure detector has several noteworthy properties. (a) It is stronger than  $\Sigma_x$  which has been shown to be necessary. (b) It is equivalent to the pair  $\langle \Sigma, \Omega \rangle$  when  $x = y = 1$  (from which it follows that  $\Pi\Sigma_{1,1}$  is optimal to solve consensus). (c) It is equivalent to the pair  $\langle \Sigma_{n-1}, \Omega_{n-1} \rangle$  when  $x = y = n - 1$  (from which it follows that  $\Pi\Sigma_{n-1,n-1}$  is optimal for  $(n - 1)$ -set agreement). (d) It is strictly weaker than the pair  $\langle \Sigma_x, \overline{\Omega}_y \rangle$  (which has been investigated in previous works) for the pairs  $(x, y)$  such that  $1 < y < x < n$ . (e) It is operational: the paper presents a  $\Pi\Sigma_{x,y}$ -based algorithm that solves  $k$ -set agreement for  $k \geq xy$ .

The second contribution of the paper is a proof that, for  $1 < k < n - 1$ , the eventual leaders failure detector  $\Omega_k$  (which eventually provides each process with the same set of  $k$  process identities, this set including at least one correct process) is not necessary to solve  $k$ -set agreement problem. More precisely, the paper shows that the weakest failure detector for  $k$ -set agreement and  $\Omega_k$  cannot be compared.

**Key-words:** Asynchronous system, Distributed computing, Eventual leader, Failure detector, Fault-tolerance, Message-passing system, Quorum, Reduction,  $k$ -Set agreement, Wait-freedom.

---

### *En quête du détecteur de fautes minimal pour le problème d'accord ensembliste*

**Résumé :** Ce rapport est une avancée dans la recherche du détecteur de fautes minimal permettant de résoudre le problème d'accord ensembliste dans un système asynchrone à communication par messages.

**Mots clés :** Calcul distribué, détecteur de fautes,  $k$ -accord, leader inéluctable, réduction, système asynchrone, tolérance aux fautes, wait-free.

---

<sup>\*</sup> IRISA, Université de Rennes, 35042 Rennes Cedex, France

<sup>\*\*</sup> Insitut Universitaire de France

<sup>\*\*\*</sup> IRISA, Université de Rennes, 35042 Rennes Cedex, France

# 1 Introduction

**The  $k$ -set agreement problem** This problem is a natural generalization of the consensus problem. It is a coordination problem (also called decision problem) introduced by S. Chaudhuri [10] to explore the relation linking the number of process failures and the minimal number of values that processes are allowed to decide. This problem can be defined as follows [10, 23]. Each process proposes a value and every non-faulty process has to decide a value (termination), in such a way that a decided value is a proposed value (validity) and no more than  $k$  different values are decided (agreement). The problem parameter  $k$  defines the coordination degree:  $k = 1$  corresponds to its most constrained instance (consensus) while  $k = n - 1$  corresponds to its weakest non-trivial instance (called set agreement).

Let  $t$  be the model parameter that defines the upper bound on the number of processes that may crash in a run,  $0 \leq t < n$ . If  $t < k$ ,  $k$ -set agreement can be trivially solved in both synchronous and asynchronous systems:  $k$  predetermined processes broadcast (write in the shared memory) the values they propose and a process decides the first proposed value it receives (reads from the shared memory). Hence, the interesting setting is when  $t \geq k$ , i.e., when the number of values that can be decided is smaller or equal to the maximal number of processes that may crash in a run.

Round-based algorithms that solve the  $k$ -set agreement problem for  $k \leq t < n$  in crash-prone synchronous message-passing systems are presented in [2, 17, 26]. These algorithms are optimal in the sense that the processes decide in at most  $\lfloor \frac{t}{k} \rfloor + 1$  rounds which has been shown to be a lower bound on the number of rounds for a process to decide<sup>1</sup>. For asynchronous systems where the processes communicate by reading/writing a shared memory or sending/receiving messages, the situation is different, namely, when  $t \geq k$ , the  $k$ -set agreement problem has no solution [6, 15, 29].

**Failure detectors** Let us observe that in an asynchronous system where the only means for processes to communicate is a read/write shared memory or send/receive message-passing network, no process is able to know if another process has crashed or is only very slow. The concept of a failure detector originates from this simple observation. A failure detector is a device (distributed oracle) that enriches a distributed system by providing alive processes with information on failed processes [8]. Several classes of failure detectors can be defined according to the type of information on failures they provide to processes (see [24] for an introduction to failure detectors).

Given a system model  $\mathcal{M}$  (e.g., asynchronous read/write shared memory system model or asynchronous send/receive message-passing system model) a failure detector  $A$  is stronger than a failure detector  $B$  with respect to  $\mathcal{M}$  (denoted  $A \succeq_{\mathcal{M}} B$  or  $B \preceq_{\mathcal{M}} A$ ) if there is an algorithm (called *reduction*) that builds  $B$  in  $\mathcal{M}$  enriched with  $A$  (we then also say that  $B$  is weaker than  $A$ ). If  $A$  is stronger than  $B$  and  $B$  is stronger than  $A$ , then  $A$  and  $B$  are equivalent with respect to  $\mathcal{M}$  (denoted  $A \simeq_{\mathcal{M}} B$ ). If  $A \succeq_{\mathcal{M}} B$  and  $B \not\preceq_{\mathcal{M}} A$  then  $A$  is strictly stronger than  $B$  -equivalently  $B$  is strictly weaker than  $A$ - (denoted  $A \succ_{\mathcal{M}} B$  or  $B \prec_{\mathcal{M}} A$ ). If  $A \not\preceq_{\mathcal{M}} B$  and  $B \not\preceq_{\mathcal{M}} A$  (denoted  $A \not\preceq_{\mathcal{M}} B$ ),  $A$  and  $B$  cannot be compared in  $\mathcal{M}$ .

Failure detectors have been investigated since 2000 [18] to circumvent the “ $t \geq k$ ” impossibility result associated with the  $k$ -set agreement problem in asynchronous systems. (Random oracles to solve the  $k$ -set agreement problem have also been investigated [19].) The question of the weakest failure detector to solve the  $k$ -set agreement problem ( $k > 1$ ) has been stated first in [28]. A failure detector  $A$  is the weakest failure detector that allows a problem  $P$  to be solved in a model  $\mathcal{M}$  if any failure detector  $B$  that allows  $P$  to be solved in  $\mathcal{M}$  is such that  $B \succeq_{\mathcal{M}} A$ .

**The weakest failure detector for  $k$ -set agreement in shared memory systems where  $t = n - 1$**  The *eventual leader* failure detector  $\Omega$  introduced in [9] is the weakest failure detector that allows consensus (i.e., 1-set agreement) to be solved in shared memory systems where any number of processes may crash [16].  $\Omega$  ensures that there is an unknown but finite time after which all the processes have the same non-faulty leader (before that time, there is an anarchy period during which each process can have an arbitrarily changing faulty or non-faulty leader). At the other end of the spectrum ( $k = n - 1$ ), the failure detector  $\overline{\Omega}_{n-1}$  (anti-omega) has been introduced in [30] where it is shown to be the weakest failure detector that allows  $(n - 1)$ -set agreement to be solved in these systems.

A simple generalization of  $\Omega$  and  $\overline{\Omega}_{n-1}$  denoted  $\overline{\Omega}_k$ ,  $1 \leq k \leq n - 1$ , ( $\overline{\Omega}_1$  is  $\Omega$ ) has been introduced in [22] where it is conjectured to be the weakest failure detector class for solving  $k$ -set agreement in asynchronous read/write shared memory systems. This conjecture has been proved in [13]. A failure detector of the class  $\overline{\Omega}_k$  provides each process with a (possibly always changing) set of  $k$  processes such that, after some unknown but finite time, all the sets that are output have in common the same non-faulty process. The optimality of  $\overline{\Omega}_k$  to solve  $k$ -set agreement in shared memory systems seems to be related to the fact that this problem is equivalent to the  $k$ -simultaneous consensus problem [1] in which each process executes  $k$  independent consensus instances (to which it proposes the same input value) and is required to terminate in one of them. As indicated in [30], this problem has been instrumental in determining the weakest failure detector for wait-free solving the  $(n - 1)$ -set agreement problem in asynchronous shared memory systems.

**The cases  $k = 1$  and  $k = n - 1$  in message-passing systems where  $t = n - 1$**  When  $k = 1$ , as already indicated  $k$ -set agreement boils down to consensus, and it is known that the failure detector denoted  $\Omega$  is the weakest to solve consensus in asynchronous message-passing systems where  $t < n/2$  [9]. This lower bound result is extended to any value of  $t$  in [11] where the failure detector  $\Sigma$  is introduced and

<sup>1</sup>In synchronous systems with more severe failures such as general omission failures,  $\lfloor \frac{t}{k} \rfloor + 1$  is still an upper bound on the number of rounds but  $k$ -set agreement can be solved if and only if  $t < \frac{k \cdot n}{k+1}$  [26].

is shown that  $\Sigma \times \Omega$  is the weakest failure detector to solve consensus in message-passing systems when  $t < n$ . This means that  $\Sigma$  is the minimal additional power (as far as information on failures is concerned) required to overcome the barrier  $t < n/2$  and attain  $t \leq n - 1$ . Actually the power provided by  $\Sigma$  is the minimal one required to implement a shared register in a message-passing system [4, 11].  $\Sigma$  provides each process with a quorum (set of process identities) such that the values of any two quorums (each taken at any time) intersect, and there is a finite time after which any quorum includes only correct processes. Fundamentally,  $\Sigma$  prevents partitioning. A failure detector  $\Sigma \times \Omega$  outputs a pair of values, one for  $\Sigma$  and one for  $\Omega$ .

The *Loneliness* failure detector (denoted  $\mathcal{L}$ ) has been proposed in [12] where it is proved that it is the weakest failure detector for solving  $(n - 1)$ -set agreement in the asynchronous message-passing model with  $t = n - 1$ . Such a failure detector provides each process  $p$  with a boolean (that  $p$  can only read) such that the boolean of at least one process remains always false and, if all but one process crash, the boolean of the remaining process becomes and remains true forever. Let us notice that the weakest failure detector for  $(n - 1)$ -set agreement is not the same in the read/write shared memory model (where it is  $\bar{\Omega}_{n-1}$ ) and the send/receive message-passing model (where it is  $\mathcal{L}$ ).

**The quest for the weakest failure detector for  $k$ -set agreement in message-passing systems** This quest seems to be one of the most difficult research topics in the theory of fault-tolerant distributed computing. Since a few years, several new failure detectors have been proposed for solving  $k$ -set agreement in asynchronous message passing systems prone to any number of crashes, but so far finding the weakest still remains a challenge.

A recent survey on failure detectors proposed so far to solve  $k$ -set agreement has appeared in [27]. The interested reader will also find in [20] a study on relations linking some of these failure detectors. Here we only present the failure detector  $\Sigma_x$  introduced in [5] because it is central to the paper.  $\Sigma_x$  generalizes the quorum failure detector class  $\Sigma$  introduced in [11] ( $\Sigma_1$  is  $\Sigma$ ). This failure detector provides each process with a set (quorum) such that at least two quorums do intersect in any set of  $x + 1$  quorums (whose values are taken at any times). Moreover, there is a finite (but unknown) time after which the quorum of any process includes only non-faulty processes. Two main results are proved in [5]: (a) as far as information on failures is concerned,  $\Sigma_k$  is a necessary requirement to solve  $k$ -set agreement; (b)  $\Sigma_{n-1}$  is sufficient to solve  $(n - 1)$ -set agreement. Interestingly, a  $\Sigma_x$ -based algorithm is presented in [7] that solves  $k$ -set agreement for  $k \geq n - \lfloor \frac{n}{x+1} \rfloor$ .

**Contributions of the paper** This paper is a new step in the quest for the weakest failure detector for  $k$ -set agreement in message-passing systems. It has two main contributions.

- The first contribution is the definition and the investigation of a new failure detector class denoted  $\Pi\Sigma_{x,y}$ .
  - Intuitively  $\Pi\Sigma_{x,1}$  (1) prevents the system from partitioning into more than  $k$  independent subsets and (2) guarantees that the processes of at least one of these subsets agree on a common leader.  $\Pi\Sigma_{x,y}$  can be seen as  $y$  independent instances of  $\Pi\Sigma_{x,1}$  in which item (2) is has to be guaranteed in only one of these instances.
  - Let  $\mathcal{AMP}$  denote the asynchronous message-passing system model where up to  $n - 1$  process may crash. The properties of  $\Pi\Sigma_{x,y}$  are the following: (a)  $\Pi\Sigma_{1,y} \simeq_{\mathcal{AMP}} \langle \Sigma_1, \bar{\Omega}_y \rangle$ ; (b)  $\Pi\Sigma_{x,n-1} \simeq_{\mathcal{AMP}} \Sigma_x$ ; (c)  $\Pi\Sigma_{x,y} \preceq_{\mathcal{AMP}} \langle \Sigma_x, \bar{\Omega}_y \rangle$  for  $1 \leq x, y \leq n$  and  $\Pi\Sigma_{x,y} \prec_{\mathcal{AMP}} \langle \Sigma_x, \bar{\Omega}_y \rangle$  for  $1 < y < x < n$ .

It follows from (a) and (b) that  $\Pi\Sigma_{1,1}$  and  $\Pi\Sigma_{n-1,n-1}$  are the weakest failure detectors to solve  $k$ -set agreement for  $k = 1$  and  $k = n - 1$ , respectively.

For  $1 \leq k \leq n - 1$ , we have the following. An algorithm based on the pair of failure detectors  $\langle \Sigma_x, \bar{\Omega}_y \rangle$  is presented in [7] that solves  $k$ -set agreement for  $k \geq xy$  (let BT-2010 denote this algorithm). Moreover, it is shown in that paper that there is no  $\langle \Sigma_x, \bar{\Omega}_y \rangle$ -based  $k$ -set agreement algorithm when  $(k < xy) \wedge (n \geq 2xy)$ .

Actually, an appropriate modification of BT-2010 provides us with a  $\Pi\Sigma_{x,y}$ -based  $k$ -set algorithm that has the same properties (listed above) as BT-2010. The important point is here the following one: while BT-2010 and the proposed algorithm work for the same pairs  $(x, y)$ , it follows from item (c) that the proposed algorithm is based on weaker information on failures than BT-2010.

- The second contribution of the paper (which has been obtained thanks to the previous failure detector  $\Pi\Sigma_{x,y}$ ) is the following:  $\Omega_k$  is not necessary to solve  $k$ -set agreement when  $1 < k < n - 1$ . Combined with the fact that  $\Sigma_k$  is necessary [5], this result restricts the area we have to look for in order to discover the weakest failure detector for  $k$ -set agreement in message-passing systems for  $1 < k < n - 1$ .

**Roadmap** The paper is made up of 7 sections. Section 2 presents the base computation model and the  $k$ -set agreement problem. Section 3 presents the eventual leaders and generalized quorums failure detectors. Section 4 defines the new failure detector  $\Pi\Sigma_{x,y}$  and shows that it is strictly weaker than  $\langle \Sigma_x, \bar{\Omega}_y \rangle$  for  $1 < y < x < n$ . Assuming  $k \geq xy$ , Section 5 presents an algorithm that solves the  $k$ -set agreement problem in the asynchronous message-passing model enriched with  $\Pi\Sigma_{x,y}$ . Section 6 shows that  $\Omega_k$  is not necessary for solving  $k$ -set agreement (when  $1 < k < n - 1$ ). Finally, Section 7 concludes the paper.

## 2 Base computation model and $k$ -set agreement

### 2.1 Computation model

**Process model** The system consists of a set of  $n$  sequential processes denoted  $p_1, \dots, p_n$ .  $\mathcal{P} = \{1, \dots, n\}$  is the set of process identities. Each process executes a sequence of (internal or communication) atomic steps. A process executes its code until it possibly crashes (if it ever crashes). After it has crashed, a process executes no more steps. A process that crashes in a run is said *faulty* in that run, otherwise it is *correct*. Given a run,  $\mathcal{C}$  and  $\mathcal{F}$  denote the set processes that are correct and the set of processes that are faulty in that run, respectively. Up to  $t = n - 1$  processes may crash in a run, hence,  $1 \leq |\mathcal{C}| \leq n$ .

**Communication model** The processes communicate by executing atomic communication steps which are the sending or the reception of a message. Every pair of processes is connected by a bidirectional channel. The channels are failure-free (no creation, alteration, duplication or loss of messages) and asynchronous (albeit the time taken by a message to travel from its sender to its receiver is finite, there is no bound on transfer delays). The notation “broadcast MSG\_TYPE( $m$ )” is used as a (non-atomic) shortcut for “**for each**  $j \in \mathcal{P}$  **do** send MSG\_TYPE( $m$ ) **to**  $p_j$  **end for**” (let us observe that  $p_i$  sends then the message also to itself).

**Underlying time model** The underlying *time model* is the set  $\mathbb{N}$  of natural integers. As we are in an asynchronous system, this time notion is not accessible to the processes (hence, the model is sometimes called time-free model). It can only be used from an external observer point of view to state or prove properties. Time instants are denoted  $\tau, \tau'$ , etc.

**Notation** The previous asynchronous crash-prone message-passing system model is denoted  $\mathcal{AMP}[\emptyset]$ .  $\mathcal{AMP}$  stands for “Asynchronous Message-Passing”;  $\emptyset$  means this is the “base” system (not enriched with a failure detector).

### 2.2 The $k$ -Set agreement problem

As already indicated in the Introduction, the  $k$ -set agreement problem has been introduced by Soma Chaudhuri [10]. It generalizes the consensus problem (that corresponds to  $k = 1$ ). It is defined as follows. Each process proposes a value and has to decide a value in such a way that the following properties are satisfied:

- Termination. Every correct process decides a value.
- Validity. A decided value is a proposed value.
- Agreement. At most  $k$  different values are decided.

## 3 Existing families of failure detectors

This section presents failure detectors that have been proposed in the quest of the weakest failure detector for  $k$ -set agreement. The system model  $\mathcal{AMP}[\emptyset]$  enriched with a failure detector  $A$  is denoted  $\mathcal{AMP}[A]$ .

A failure detector provides each alive process with a read-only local variable. Let  $xxx_i$  be such a variable of process  $p_i$ . Let  $xxx_i^\tau$  denotes the value of  $xxx_i$  at time  $\tau$ .

### 3.1 The $\Omega_k$ and $\overline{\Omega}_k$ families

The *eventual leaders* failure detectors of the families  $\Omega_k$  and  $\overline{\Omega}_k$  provide each process  $p_i$  with a local variable denoted  $leaders_i$ . They originates from  $\Omega$  [9] ( $\Omega_1 \equiv \overline{\Omega}_1 \equiv \Omega$ ).  $\overline{\Omega}_k$  is a straightforward generalization of  $\overline{\Omega}_{n-1}$  (introduced in [30]).  $\overline{\Omega}_k$  has been shown to be the weakest failure detector to solve  $k$ -set agreement in asynchronous shared memory systems with any number of process crashes in [13].

**Base properties** Let us consider the following properties on the sets  $leaders_i$ .

- Validity.  $\forall i, \forall \tau$ :  $leaders_i^\tau$  is a set of  $k$  process identities.
- Strong eventual leadership.  $\exists LD, \tau$ :  $(LD \cap \mathcal{C} \neq \emptyset) \wedge (\forall \tau' \geq \tau, \forall i \in \mathcal{C} : leaders_i^{\tau'} = LD)$ .
- Weak eventual leadership.  $\exists \ell \in \mathcal{C}, \tau$ :  $(\forall \tau' \geq \tau, \forall i \in \mathcal{C} : \ell \in leaders_i^{\tau'})$ .

Validity combined with strong eventual leadership states that, after some unknown but finite time, all correct processes have the same set of  $k$  leaders and at least one of them is a correct process. Validity combined with weak eventual leadership requires only that the correct processes eventually share a common correct leader.

**The  $\Omega_k$  family** This family (introduced in [21]) includes the failure detectors that satisfy the validity and strong eventual leadership properties.

**The  $\overline{\Omega}_k$  family** This family (introduced in [22]) includes the failure detectors that satisfy the validity and weak eventual leadership properties.

### 3.2 The $\Sigma_k$ and $\Pi_k$ families

**The  $\Sigma_k$  family** As noticed in the Introduction, the *generalized quorum* failure detector  $\Sigma_k$  (introduced in [5]) is a generalization of the quorum failure detector  $\Sigma$  introduced in [11] where it is shown to be the weakest failure detector to implement a register in  $\mathcal{AMP}[\emptyset]$ .

$\Sigma_k$  provides each process  $p_i$  with a set  $qr_i$  (called quorum) that satisfies the following properties (after a process  $p_i$  has crashed, we have  $qr_i = \mathcal{P}$  by definition). The self-inclusion property (which does not appear in [5]) is considered here because it allows for a simpler formulation of algorithms.

- Self-inclusion.  $\forall i \in \mathcal{P}, \forall \tau: i \in qr_i^\tau$ .
- Quorum liveness.  $\exists \tau: \forall i \in \mathcal{C}, \forall \tau' \geq \tau: qr_i^{\tau'} \subseteq \mathcal{C}$ .
- Quorum intersection.  $\forall id_1, \dots, id_{x+1} \in \mathcal{P}, \forall \tau_1, \dots, \tau_{x+1}: \exists i, j: (i \neq j) \wedge (qr_{id_i}^{\tau_i} \cap qr_{id_j}^{\tau_j} \neq \emptyset)$ .

It is shown in [5] that  $\Sigma_k$  is necessary when one wants to solve  $k$ -set agreement in  $\mathcal{AMP}[\emptyset]$ .

**The  $\Pi_k$  family** The failure detector  $\Pi_k$  (introduced in [5]) is an extension of  $\Sigma_k$  to which it adds the following property.

- Eventual leadership.  $\exists LD, \tau: (|LD| = k) \wedge (\forall \tau' \geq \tau, \forall i \in \mathcal{C}: qr_i^{\tau'} \cap LD \neq \emptyset)$ .

It is shown in [5] that  $\Pi_k$  and the pair  $\langle \Sigma_k, \Omega_k \rangle$  are equivalent, i.e.,  $\Pi_k$  can be built in  $\mathcal{AMP}[\Sigma_k, \Omega_k]$  and  $\langle \Sigma_k, \Omega_k \rangle$  can be built in  $\mathcal{AMP}[\Pi_k]$ . It is also shown in [5] that  $\Pi_{n-1}$  and  $\mathcal{L}$  are equivalent. It follows from these observations that  $\Pi_1$  and  $\Pi_{n-1}$  are the weakest failure detectors for  $k = 1$  and  $k = n - 1$ . Unfortunately, as shown in [3, 7],  $\Pi_k$  does not allow to solve  $k$ -set agreement for  $1 < k < n - 1$ .

## 4 The family of failure detectors $\Pi\Sigma_{x,y}$

### 4.1 Definition

The definition of  $\Pi\Sigma_{x,y}$  is incremental, first is defined  $\Pi\Sigma_x$  and then  $\Pi\Sigma_{x,y}$ .

**The failure detector  $\Pi\Sigma_x$**  A failure detector  $\Pi\Sigma_x$  provides each process  $p_i$  with a set  $qr_i$  and a variable  $leader_i$  which define the current quorum and the current leader of  $p_i$ . It is defined by the following properties.

- Self-inclusion.  $\forall i \in \mathcal{P}, \forall \tau: i \in qr_i^\tau$ .
- Quorum liveness.  $\exists \tau: \forall i \in \mathcal{C}, \forall \tau' \geq \tau: qr_i^{\tau'} \subseteq \mathcal{C}$ .
- Quorum intersection.  $\forall id_1, \dots, id_{x+1} \in \mathcal{P}, \forall \tau_1, \dots, \tau_{x+1}: \exists i, j: (i \neq j) \wedge (qr_{id_i}^{\tau_i} \cap qr_{id_j}^{\tau_j} \neq \emptyset)$ .
- Eventual partial leadership.  $\exists \ell \in \mathcal{C}: \forall i \in \mathcal{C}: (\forall \tau: \exists \tau_i, \tau_\ell \geq \tau: qr_i^{\tau_i} \cap qr_\ell^{\tau_\ell} \neq \emptyset) \Rightarrow (\exists \tau: \forall \tau' \geq \tau: leader_i^{\tau'} = \ell)$ .

The self-inclusion, liveness and intersection properties are the properties that define  $\Sigma_x$ : after some time the quorum of any correct process contains only correct processes (liveness) and any set of  $x + 1$  quorums contains two intersecting quorums (intersection). Hence,  $\Pi\Sigma_x \succeq \Sigma_x$ .

Eventual partial leadership states that there is a correct process  $p_\ell$  such that, for any correct process  $p_i$  whose quorum  $qr_i$  intersects infinitely often its quorum  $qr_\ell$  (left part of the implication), then eventually  $p_\ell$  is forever the leader of  $p_i$  (right part of the implication).

**The failure detector  $\Pi\Sigma_{x,y}$**   $\Pi\Sigma_x$  is  $\Pi\Sigma_{x,1}$ . More generally,  $\Pi\Sigma_{x,y}$  provides each process  $p_i$  with an array  $FD_i[1..y]$  such that for each each  $j$ ,  $1 \leq j \leq y$ ,  $FD_i[j]$  is a pair containing a quorum  $FD_i[j].qr$  and a process index  $FD_i[j].leader$ . Let  $FD[j]$  denote the corresponding distributed object. The failure detector  $\Pi\Sigma_{x,y}$  consists of an array  $FD[1..y]$  that satisfies the following properties:

- Vector safety.  $\forall j \in [1..y]: FD[j].qr$  satisfies the liveness and intersection properties of  $\Pi\Sigma_x$ .
- Vector liveness.  $\exists j \in [1..y]: FD[j]$  satisfies the eventual partial leadership property of  $\Pi\Sigma_x$ .

## 4.2 $\Pi\Sigma_{x,y}$ vs $\langle \Sigma_x, \bar{\Omega}_y \rangle$

A failure detector  $\langle \Sigma_x, \bar{\Omega}_y \rangle$  provides each process  $p_i$  with two *independent* read-only local variables:  $qr_i$  that satisfies the properties defined by  $\Sigma_x$ , and  $leaders_i$  that satisfies the properties defined by  $\bar{\Omega}_y$ .

**Lemma 1.** *Let  $1 \leq x, y \leq n - 1$ .  $\Pi\Sigma_{x,y} \preceq_{\mathcal{AMP}} \langle \Sigma_x, \bar{\Omega}_y \rangle$ .*

**Proof** Let  $qr_i$  be the output of  $\Sigma_x$  at process  $p_i$ . For any  $j \in [1..y]$ , let  $FD_i[j].qr = qr_i$ . Hence, each quorum  $FD_i[j].qr$  inherits from the liveness and intersection properties of  $\Sigma_x$  and consequently the vector safety property of  $\Pi\Sigma_{x,y}$  is satisfied.

The proof of the vector liveness property of  $\Pi\Sigma_{x,y}$  is similar to the proof showing that vector- $\Omega$  can be built from  $\bar{\Omega}_{n-1}$  [30]. Each process  $p_i$  executes Algorithm 1 in which  $susp\_nb_i[1..n]$  is a local variable of  $p_i$  initialized to  $[0, \dots, 0]$ . Moreover, the processes are provided with a reliable broadcast operation denoted `rel_broadcast`. Such an operation ensures that, if a message is delivered by a process (which can be correct or not), then it is delivered by all correct processes. This operation can be implemented in  $\mathcal{AMP}[0]$  (e.g., [8, 25]). Each process  $p_i$  repeatedly broadcasts (with `rel_broadcast`) its current value of  $leaders_i$  (line 01). When, it is delivered `LEADER(ld)`

```

(01) repeat forever rel_broadcast LEADER(leadersi) end repeat.

(02) when LEADER(ld) is delivered:
(03)   for each  $j \notin ld$  do  $susp\_nb_i[j] \leftarrow susp\_nb_i[j] + 1$  end for;
(04)   let  $j_1, j_2, \dots, j_n$  be a permutation of  $\{1, \dots, n\}$  such that
         $(susp\_nb_i[j_1], j_1) < (susp\_nb_i[j_2], j_2) < \dots < (susp\_nb_i[j_n], j_n)$ ;
(05)   for each  $x \in \{1, \dots, y\}$  do  $FD_i[x].leader \leftarrow j_x$  end for.

```

Algorithm 1: From  $\langle \Sigma_x, \bar{\Omega}_y \rangle$  to  $\Pi\Sigma_{x,y}$  (code for  $p_i$ )

a process  $p_i$  first increases the suspicion number  $susp\_nb_i[j]$  of all  $j \notin ld$  (line 03). It then sorts process identities according to the lexicographical order on the pairs  $\{(susp\_nb_i[j], j)\}_{1 \leq j \leq n}$  (line 04). Finally, for each  $x \in [1..y]$ ,  $p_i$  assigns to  $FD_i[x].leader$  (line 05) the  $x$ th process index (as defined from the previous order).

Let us observe that, due to the weak eventual leadership property of  $\bar{\Omega}_y$ , there is a correct process  $p_\ell$  that, after some finite time, belongs permanently to all local variables  $leaders_i$ . It follows from this observation and the `rel_broadcast()` operation that all the local variables  $susp\_nb_i[\ell]$  will stop increasing and stabilize to the very same value. As  $\bar{\Omega}_y$  outputs sets of  $y$  processes, this is true for  $m$  processes  $p_{\ell_1}, p_{\ell_2}, \dots$ , with  $1 \leq m \leq y$ . It then follows from line 05 that, for all the processes  $p_i$  and for each  $z \in [1..m]$ , the local variables  $FD_i[z].leader$  stabilizes to the very same process index and there is one entry  $j$  such that we eventually have forever  $FD_i[j].leader = \ell$  such that  $p_\ell$  is a correct process. As, when considering the distributed object  $FD[j]$ , all processes have the same correct leader  $p_\ell$ , the eventual partial leadership property of  $\Pi\Sigma_x$  is satisfied (more precisely, whatever the value of the left part of the implication, the right part of the implication is satisfied). Consequently the vector liveness property of  $\Pi\Sigma_{x,y}$  is also satisfied, which concludes the proof of the lemma.  $\square_{\text{Lemma 1}}$

**Theorem 1.** *Let  $1 \leq y \leq n - 1$ .  $\Pi\Sigma_{1,y} \simeq_{\mathcal{AMP}} \langle \Sigma_1, \bar{\Omega}_y \rangle$ .*

**Proof** Taking  $x = 1$  in Lemma 1 we have  $\Pi\Sigma_{1,y} \preceq_{\mathcal{AMP}} \langle \Sigma_1, \bar{\Omega}_y \rangle$ . Hence, we have only to show that  $\langle \Sigma_1, \bar{\Omega}_y \rangle \preceq_{\mathcal{AMP}} \Pi\Sigma_{1,y}$ .

Let  $qr_i$  of  $\Sigma_1$  be the output of  $FD_i[1].qr$ . Hence, the quorums  $qr_i$  inherit the liveness and intersection properties of  $FD_i[1].qr$  that trivially satisfy the properties defining  $\Sigma_1$ .

Let  $leaders_i$  be any subset of size  $y$  that contains  $\cup_{1 \leq j \leq y} \{FD_i[j].leader\}$ . Due to the vector liveness property of  $\Pi\Sigma_{1,y}$ , there is an entry  $j$  such that  $FD[j]$  satisfies the eventual partial leadership (Observation O1). Moreover, as  $x = 1$ , any pair of quorums output by  $FD[j]$  do intersect (Observation O2). It follows from O1 and O2 that there is a correct process  $p_\ell$  such that, for each correct process  $p_i$ , there is a time after which the predicate  $FD_i[j].leader = \ell$  remains forever true. Consequently, there is a finite time after which the predicate  $\ell \in leaders_i$  remains forever true at any correct process  $p_i$ , from which follows the weak eventual leadership property of  $\bar{\Omega}_y$ .  $\square_{\text{Theorem 1}}$

**Theorem 2.** *Let  $1 \leq x \leq n - 1$ .  $\Pi\Sigma_{x,n-1} \simeq_{\mathcal{AMP}} \Sigma_x$ .*

**Proof** Taking  $y = n - 1$  in Lemma 1 we have  $\Pi\Sigma_{x,n-1} \preceq_{\mathcal{AMP}} \langle \Sigma_x, \bar{\Omega}_{n-1} \rangle$ . Hence, we have only to show that  $\langle \Sigma_x, \bar{\Omega}_{n-1} \rangle \preceq_{\mathcal{AMP}} \Pi\Sigma_{x,n-1}$ .

It is shown in [5] (Corollary 2 in [5]) that  $\Sigma_x \succeq_{\mathcal{AMP}} \Sigma_{x+1} \succeq_{\mathcal{AMP}} \dots \succeq_{\mathcal{AMP}} \Sigma_{n-1} \succeq_{\mathcal{AMP}} \Omega_{n-1}$ . Moreover, it follows directly from their definitions that  $\Omega_{n-1} \succeq_{\mathcal{AMP}} \bar{\Omega}_{n-1}$ . It follows that  $\Sigma_x \succeq_{\mathcal{AMP}} \langle \Sigma_x, \bar{\Omega}_{n-1} \rangle$  which completes the proof of the theorem.  $\square_{\text{Theorem 2}}$

**Lemma 2.** *Let  $1 \leq y < n$ .  $\Pi\Sigma_{y+1,1} \not\succeq_{\mathcal{AMP}} \bar{\Omega}_y$ .*

The proof of this lemma is given in Appendix A.

**Theorem 3.** Let  $1 < y < x < n$ .  $\Pi\Sigma_{x,y} \prec_{\mathcal{AMP}} \langle \Sigma_x, \bar{\Omega}_y \rangle$ .

**Proof**  $\Pi\Sigma_{x,y} \preceq_{\mathcal{AMP}} \langle \Sigma_x, \bar{\Omega}_y \rangle$  follows from Lemma 1. Hence, we have to show that  $\Pi\Sigma_{x,y} \not\prec_{\mathcal{AMP}} \langle \Sigma_x, \bar{\Omega}_y \rangle$ . Let us first observe that  $\Pi\Sigma_{y+1,1} \succeq_{\mathcal{AMP}} \Pi\Sigma_{y+1,y}$ . This is easily obtained by providing each  $FD[j]$  of the array  $FD[1..y]$  of  $\Pi\Sigma_{y+1,y}$  with the outputs supplied by  $\Pi\Sigma_{y+1,1}$ . On an other side, (as shown in [5])  $\Sigma_z$  is strictly stronger than  $\Sigma_{z+1}$  for  $1 \leq z < n-1$  and, consequently,  $\Pi\Sigma_{y+1,y} \succeq_{\mathcal{AMP}} \Pi\Sigma_{x,y}$  for  $y < x$ . It follows from Lemma 2 (i.e.,  $\Pi\Sigma_{y+1,1} \not\prec_{\mathcal{AMP}} \bar{\Omega}_y$ ) that, as  $\Pi\Sigma_{y+1,1}$  is stronger than  $\Pi\Sigma_{x,y}$ , we have  $\Pi\Sigma_{x,y} \not\prec_{\mathcal{AMP}} \bar{\Omega}_y$  which proves the theorem.  $\square_{Theorem 3}$

**Remark** A main difference between  $\Pi\Sigma_{x,y}$  and  $\langle \Sigma_x, \bar{\Omega}_y \rangle$  lies in the fact that the eventual correct leader elected by  $\bar{\Omega}_y$  has to be the same for all correct processes, while  $\Pi\Sigma_{x,y}$  requires only that the correct processes of a subset (dynamically defined by one of the  $y$   $\Sigma_x$  failure detectors) agree on a common leader. Hence, the scope of the leadership provided by  $\Pi\Sigma_{x,y}$  is not required to be the whole system but only a subset of it.

## 5 Solving $k$ -Set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,y}]$

This paper presents an algorithm that solves the  $k$ -set agreement problem in  $\mathcal{AMP}[\Pi\Sigma_{x,y}]$ . This algorithm is similar to the one presented in [7] which in turn is an adaptation of the algorithms described in [14, 28].

### 5.1 The $\text{Alpha}_x$ abstraction

This abstraction has been introduced in [14] to capture the safety property of consensus and generalized in [28] to capture the safety property of  $k$ -set agreement in crash-prone systems. Corresponding implementations in read/write shared memory systems and send/receive message-passing systems can be found in [14, 28].

Let  $\perp$  be a default value that cannot be proposed by processes.  $\text{Alpha}_x$  is an object initialized to  $\perp$  that may store up to  $x$  different values proposed by processes. It is an abstraction (object) that provides processes with a single operation denoted  $\text{propose}(r, v)$  (where  $r$  is a round number and  $v$  a proposed value) that returns a value to the invoking process. The round number plays the role of a logical time that allows identifying the  $\text{propose}()$  invocations. It is assumed that distinct processes use different round numbers and successive invocations by the same process use increasing sequence numbers.  $\text{Alpha}_x$  is a kind of *abortable* object in the sense that  $\text{propose}()$  invocations are allowed to return the default value  $\perp$  (i.e., abort) in specific concurrency-related circumstances (as defined from the obligation property, see below). More precisely, the  $\text{Alpha}_x$  objects used in this paper are defined by the following specification in which the obligation property takes explicitly into account the fact that we are interested into an  $\text{Alpha}_x$  object that will be implemented on top of  $\mathcal{AMP}[\Sigma_x]$  (which is a strictly stronger underlying model than  $\mathcal{AMP}[\emptyset]$ ).

- Termination. Any invocation of  $\text{propose}()$  by a correct process terminates.
- Validity. If  $\text{propose}(r, v)$  returns  $v' \neq \perp$ , then  $\text{propose}(r', v')$  has been invoked with  $r' \leq r$ .
- Quasi-agreement. At most  $k$  different non- $\perp$  values can be returned by  $\text{propose}()$  invocations.
- Obligation.  $p_\ell$  being a correct process let  $Q(\ell, \tau) = \{i \in \mathcal{C} \mid \forall \tau_i, \tau_\ell \geq \tau : qr_i^{\tau_i} \cap qr_\ell^{\tau_\ell} = \emptyset\}$ . If, after time  $\tau$ , (a) only  $p_\ell$  and processes of  $Q(\ell, \tau)$  invoke  $\text{propose}()$  and (b)  $p_\ell$  invokes  $\text{propose}()$  infinitely often, then at least one invocation issued by  $p_\ell$  returns a non- $\perp$  value.

Differently from the obligation property stated in [7, 14, 28] the previous obligation property is  $\Sigma_x$ -aware which allows for a weaker property (the  $\text{Alpha}_x$  object used in [7] is implemented on top of  $\mathcal{AMP}[\Sigma_x]$  but its specification is not  $\Sigma_x$ -aware). More precisely, our obligation property allows concurrent invocations of  $\text{propose}()$  to return non- $\perp$  values as soon as the quorums of the invoking processes do not intersect during these invocations.

An algorithm implementing the previous  $\text{Alpha}_x$  object in  $\mathcal{AMP}[\Sigma_x]$  is described in Appendix B.

### 5.2 $k$ -set agreement in $\mathcal{AMP}[\Pi\Sigma_{x,y}]$

This section presents a simple algorithm that implements  $k$ -set agreement in  $\mathcal{AMP}[\Pi\Sigma_{x,y}]$  for  $k \geq xy$ . It is as the algorithm presented in [7]: it uses a base algorithm (similar to the one introduced in [14]) that solves  $x$ -set agreement in  $\mathcal{AMP}[\Pi\Sigma_{x,1}]$  and then assuming  $k \geq xy$  (as in [1, 7]) it uses  $y$  instances of this base to solve  $k$ -set agreement in  $\mathcal{AMP}[\Pi\Sigma_{x,y}]$ .



**$x$ -Set agreement in  $\mathcal{AMP}[\Pi\Sigma_{x,1}]$**  Algorithm 2 solves  $x$ -set agreement in  $\mathcal{AMP}[\Pi\Sigma_{x,1}]$ . A process  $p_i$  invokes  $\text{ks\_propose}_{x,1}(v_i)$  where  $v_i$  is the value it proposes. It decides a value  $d$  when it executes the statement  $\text{return}(d)$  which terminates its invocation. The local variable  $r_i$  is the local round number (as it is easy to see, each process uses increasing round numbers and no two distinct processes use the same round numbers).

A process loops until it decides. If during a loop iteration  $p_i$  is such that  $\text{leader}_i = i$  ( $\text{leader}_i$  is one of the two local outputs provided by  $\Pi\Sigma_{x,1}$ ),  $p_i$  invokes the  $\text{Alpha}_x$  object to try to deposit its value  $v_i$  into it (the success depends on the concurrency and quorums pattern). If a non- $\perp$  value is returned by this invocation,  $p_i$  broadcasts it (with the reliable broadcast operation). A process decides as soon as it is delivered a  $\text{DECISION}()$  message.

```

operation  $\text{ks\_propose}_{x,1}(v_i)$ :
(01)  $\text{dec}_i \leftarrow \perp$ ;  $r_i \leftarrow i$ ;
(02) while ( $\text{dec}_i = \perp$ ) do
(03)   if ( $\text{leader}_i = i$ ) then  $\text{Alpha}_x.\text{propose}(r_i, v_i)$ ;  $r_i \leftarrow r_i + n$  end if
(04) end while;
(05)  $\text{rel\_broadcast DECISION}(\text{dec}_i)$ .

when  $\text{DECISION}(d)$  is delivered:  $\text{return}(d)$ .

```

Algorithm 2:  $x$ -Set agreement in  $\mathcal{AMP}[\Pi\Sigma_{x,1}]$  (code for  $p_i$ )

**Theorem 4.** Algorithm 2 solves the  $x$ -set agreement in  $\mathcal{AMP}[\Pi\Sigma_{x,1}]$ .

The proof of this theorem is given in Appendix C.

**$k$ -Set agreement in  $\mathcal{AMP}[\Pi\Sigma_{x,y}]$**  As in [1, 7], a simple  $k$ -set algorithm can be obtained by launching concurrently  $y$  instances of Algorithm 2, the  $j$ th one relying on the component  $FD[j]$  of the failure detector  $\mathcal{AMP}[\Pi\Sigma_{x,y}]$ . A process decides the value returned by the first of the  $y$  instances that locally terminates. As there are  $y$  instances of Algorithm 2 and at most  $x$  values can be decided in each of them, it follows that at most  $xy$  different values can be decided. Moreover, as at least one  $FD[j]$  is a  $\Pi\Sigma_{x,y}$  failure detector, it follows that the correct processes decide (if not done before) in at least one of the  $y$  instances of Algorithm 2. Let us observe that, in such a “worst” case where the processes decide in the same instance, at most  $x$  values are decided).

## 6 $\Omega_k$ is not necessary for $k$ -set agreement when $1 < k < n - 1$

**Lemma 3.** Let  $1 < k < n - 1$ .  $\Omega_k$  cannot be built in  $\mathcal{AMP}[\Pi\Sigma_{k,1}]$ .

**Proof Preliminaries.** Let us first observe that  $(1 < k < n - 1) \Leftrightarrow (n \geq k + 2 \geq 4)$ . The proof is by contradiction. It consists in building distinct runs that are indistinguishable for some processes. The impossibility will follow from the fact that, while each of these runs is provided with correct outputs from  $\Pi\Sigma_{k,1}$ , there are runs in which  $\Omega_k$  has to provide each process with a set containing infinitely often  $k + 1$  process identities.

Hence, let us assume that there is an algorithm  $A$  that builds  $\Omega_k$  in  $\mathcal{AMP}[\Pi\Sigma_{x,1}]$ . Let us remember that  $\text{leaders}_i$  denotes the output of  $\Omega_k$  while  $qr_i$  and  $ld_i$  (instead of  $\text{leader}_i$  to prevent confusion) are the output provided to  $p_i$  by  $\Pi\Sigma_{x,1}$ .

The runs considered in the proof are such that the processes  $p_1, p_2$  and  $p_3$  on one side and the processes  $p_i$  with  $i \in [4..k + 2]$  on another side play special roles. Moreover, (if it exists) each process  $p_j$  with  $j \in [k + 3..n]$  is initially crashed in all runs.

**The “ $\alpha$ ” runs.** For all  $i \in [4..k + 2]$ , let  $\alpha_i$  be a run in which all processes but  $p_i$  have initially crashed and  $p_i$  does not crash. Hence,  $\forall \tau, qr_i^\tau = \{i\}$  and  $ld_i^\tau = i$  are correct outputs of  $\Pi\Sigma_{x,1}$  in the run  $\alpha_i$ . As  $p_i$  is the only correct process in  $\alpha_i$ , it follows from the very existence of algorithm  $A$  that there is a time  $\tau_i$  such that  $\forall \tau \geq \tau_i: i \in \text{leaders}_i^\tau$ .

Let  $\alpha_{[1..3]}$  be a run in which  $p_1, p_2$  and  $p_3$  do not crash and all other processes have initially crashed. For each  $i \in [1..3]$  and any time  $\tau, qr_i^\tau = \{1, 2, 3\}$  and  $ld_i^\tau = \text{lead}_\alpha$  (where  $\text{lead}_\alpha \in \{1, 2, 3\}$ ) are correct outputs of  $\Pi\Sigma_{x,1}$  in the run  $\alpha_{[1..3]}$ . Due to the existence of  $A$ , there is a correct process  $p_{\ell\alpha}$  (with  $\ell\alpha \in [1..3]$ ) and a time  $\tau_{[1..3]}$  such that  $\forall \tau \geq \tau_{[1..3]}, \forall i \in \{1, 2, 3\}, \ell\alpha \in \text{leaders}_i^\tau$ .

Let  $\tau_\alpha = \max(\{\tau_i\}_{i \in [4..k+2]}, \tau_{[1..3]})$  and  $\alpha$  be a run in which (1) all processes  $p_i$  such that  $i > k + 2$  have initially crashed, (2) the other processes are correct, (3) all messages but the ones exchanged by  $p_1, p_2$  and  $p_3$  are delayed until  $\tau_\alpha$ , and (4), up to  $\tau_\alpha$ , the outputs of  $\Pi\Sigma_{x,1}$  are the same in  $\alpha$  and  $\alpha_{[1..3]}$  for  $p_1, p_2$  and  $p_3$  and the same in  $\alpha$  and  $\alpha_i$  for  $p_i$  with  $i \in [4..k + 2]$  (the important point to observe here is that, due to the fact that the processes  $p_j$  with  $k + 2 < j \leq n$  have initially crashed, the intersection property of  $\Pi\Sigma_{x,1}$  is satisfied). Considering the runs previously defined, we have the following:

- For any  $i \in [4..k + 2]$ ,  $p_i$  cannot distinguish  $\alpha$  and  $\alpha_i$  up to time  $\tau_\alpha$ .
- Each of  $p_1, p_2$  and  $p_3$  cannot distinguish  $\alpha$  and  $\alpha_{[1..3]}$  up to time  $\tau_\alpha$ .

It follows that, considering the outputs of algorithm  $A$  in run  $\alpha$ , we have  $\ell_\alpha \in leaders_i^{\tau_\alpha}$  for  $i \in \{1, 2, 3\}$  and  $i \in leaders_i^{\tau_\alpha}$  for  $i \in [4..k+2]$ .

*The “ $\beta$ ” runs.* Let  $\tau_\alpha^f$  be a time in run  $\alpha$  after which all messages sent before  $\tau_\alpha$  have been received. Moreover, for each  $i \in [4..k+2]$ , let  $\beta_i$  be a run in which all processes take the same steps as in run  $\alpha$  until  $\tau_\alpha^f$  and then all processes but  $p_i$  crash. Let us observe that, in the run  $\beta_i$ , correct outputs of  $\Pi\Sigma_{x,1}$  at  $p_i$  can still be  $\forall \tau: qr_i^\tau = \{i\}$  and  $ld_i^\tau = i$ .

Let  $\beta_{[1..3] \setminus \{\ell_\alpha\}}$  be the same run as  $\alpha$  until  $\tau_\alpha^f$  and where all processes but  $p_i$  such that  $i \in [1..3] \setminus \{\ell_\alpha\}$  crash just after  $\tau_\alpha^f$ . In  $\beta_{[1..3] \setminus \{\ell_\alpha\}}$ ,  $\forall \tau \geq \tau_\alpha^f$ ,  $qr_i^\tau = [1..3] \setminus \{\ell_\alpha\}$  and  $ld_i^\tau = lead_\beta$  (with  $lead_\beta \in [1..3] \setminus \{\ell_\alpha\}$ ) are correct outputs of  $\Pi\Sigma_{x,1}$  at each process  $p_i$  such that  $i \in [1..3] \setminus \{\ell_\alpha\}$ .

Due to the existence of  $A$ , there is a finite time  $\tau_\beta$  such that (a) in the run  $\beta_{[1..3] \setminus \{\ell_\alpha\}}$ , there is  $\ell_\beta$  such that  $\ell_\beta \in [1..3] \setminus \{\ell_\alpha\}$  and  $\ell_\beta \in leaders_j$  for each  $j \in [1..3] \setminus \{\ell_\alpha\}$  (let us remember that  $p_{\ell_\alpha}$  has crashed just after  $\tau_\alpha^f$ ) and (b) due to the very definition of each run  $\beta_i$  for  $i \in [4..k+2]$ , we have  $i \in leaders_i^{\tau_\beta}$  in run  $\beta_i$ .

Let  $\beta$  be a run that is the same as run  $\alpha$  until time  $\tau_\alpha^f$  and where the messages sent after  $\tau_\alpha^f$  (1) by or to the processes  $p_i$  where  $i \in [4..k+2]$  and (2) by  $p_{\ell_\alpha}$  are delayed until after  $\tau_\beta$  (this means that each process  $p_i$  where  $i \in [1..3] \setminus \{\ell_\alpha\}$  cannot distinguish if  $p_{\ell_\alpha}$  is alive or crashed; differently, as it receives their messages  $p_{\ell_\alpha}$  does not suspect these two processes).

In the run  $\beta$ , let the outputs of  $\Pi\Sigma_{x,1}$  be the same as in the run  $\alpha$  until time  $\tau_\alpha^f$ , and then forever be (1) the same as in  $\beta_{[1..3] \setminus \{\ell_\alpha\}}$  for  $p_i$  with  $i \in [1..3] \setminus \{\ell_\alpha\}$ , (2) the same as in  $\beta_i$  for  $p_i$  with  $i \in [4..k+2]$ , and (3)  $qr_{\ell_\alpha} = \{1, 2, 3\}$  and  $ld_{\ell_\alpha} = lead_\beta$  for  $p_{\ell_\alpha}$  (as before, the important point to observe here is that, due to the fact that the processes  $p_j$  with  $k+2 < j \leq n$  have initially crashed, the intersection property of  $\Pi\Sigma_{x,1}$  is satisfied). Considering the runs previously defined, we have the following:

- For any  $i \in [4..k+2]$ ,  $p_i$  cannot distinguish  $\beta$  and  $\beta_i$ .
- For each  $i \in [1..3] \setminus \{\ell_\alpha\}$ ,  $p_i$  cannot distinguish  $\beta$  and  $\beta_{[1..3] \setminus \{\ell_\alpha\}}$ .

It follows that, considering the outputs of algorithm  $A$  in the run  $\beta$ , we have  $\ell_\beta \in leaders_i^{\tau_\beta}$  for  $i \in [1..3] \setminus \{\ell_\alpha\}$  and  $i \in leaders_i^{\tau_\beta}$  for  $i \in [4..k+2]$ .

*The “ $\gamma$ ” runs.* Let  $\tau_\beta^f$  be a time in the run  $\beta$  after which all the messages sent before  $\tau_\beta$  have been received. In a similar way to the one used to build the run  $\beta$  from the run  $\alpha$ , it is possible to build a run  $\gamma$  from the run  $\beta$  in which, for each  $i \in [1..3] \setminus \{\ell_\beta\}$  there is  $\ell_\gamma \in [1..3] \setminus \{\ell_\beta\}$  such that we eventually have forever  $\ell_\gamma \in leaders_i$ .

This construction can be repeated to obtain an infinite run in which at least  $k+1$  process identities (namely, the identities  $4, \dots, k+2$ , and at least two among the identities  $\{1, 2, 3\}$ ) are infinitely often in the set variables  $leaders_i$  which contradicts the strong eventual leadership property of  $\Omega_k$  while the outputs produced by  $\Pi\Sigma_{x,1}$  at each process  $p_i$  satisfy the specification of  $\Pi\Sigma_{x,1}$  (in particular, the intersection property of  $\Pi\Sigma_{x,1}$  are satisfied in all the runs that have been built).  $\square_{\text{Lemma 3}}$

**Theorem 5.** Let  $1 < k < n - 1$ .  $\Omega_k$  is not necessary for solving  $k$ -set agreement in  $\mathcal{AMP}[\emptyset]$ .

**Proof** The theorem follows directly from Lemma 3 and the fact that  $k$ -set agreement can be solved in  $\mathcal{AMP}[\Pi\Sigma_{k,1}]$ .  $\square_{\text{Theorem 5}}$

Let  $X(k)$  denote the (still unknown) weakest failure detector such that  $k$ -set agreement can be solved in  $\mathcal{AMP}[X(k)]$ . The following corollary shows that, when  $1 < k < n - 1$ ,  $\Omega_k$  is neither necessary nor sufficient for solving the  $k$ -set agreement problem.

**Corollary 1.** Let  $1 < k < n - 1$ .  $\Omega_k \not\preceq_{\mathcal{AMP}} X(k)$ .

**Proof** The proof is by contradiction. Let us first assume that  $\Omega_k \succeq_{\mathcal{AMP}} X(k)$ . In that case  $k$ -set agreement can be solved from  $\Omega_k$ . But it is impossible to solve  $k$ -set agreement in  $\mathcal{AMP}[\langle \Sigma_k, \Omega_k \rangle]$  [3].

Let us now assume that  $X(k) \succeq_{\mathcal{AMP}} \Omega_k$ . Due to the definition of  $X(k)$ , we have  $\Pi\Sigma_{k,1} \succeq_{\mathcal{AMP}} X(k)$ , hence  $\Pi\Sigma_{k,1} \succeq_{\mathcal{AMP}} \Omega_k$  (by transitivity). But, for  $1 < k < n - 1$ , this is contradicted by Lemma 3 which has shown that  $\Pi\Sigma_{k,1} \not\preceq_{\mathcal{AMP}} \Omega_k$ . It follows that  $\Omega_k$  and  $X(k)$  cannot be compared.  $\square_{\text{Corollary 1}}$

## 7 Conclusion

As indicated in the abstract, this paper is a new step in the quest for discovering the weakest failure detector for  $k$ -set agreement in asynchronous message-passing systems prone to any number of process failures. It has presented a new failure detector denoted  $\Pi\Sigma_{x,y}$  which enjoys many noteworthy features and an associated algorithm that solves  $k$ -set agreement for  $k \geq xy$ . It has also shown that the weakest failure detector (that still remains to be discovered) and  $\Omega_k$  (a well-studied failure detector) cannot be compared.

More generally, an important issue that remains to be solved lies in capturing the “weakest” type of shared memory that has to be emulated for solving  $k$ -set agreement in asynchronous message-passing systems.

## Acknowledgments

The authors want to thank Martin Biely, Peter Robinson and Ulrich Schmid for interesting discussions on failure detectors suited to the  $k$ -set agreement problem in asynchronous message-passing systems.

## References

- [1] Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., The  $k$ -Simultaneous Consensus Problem. *Distributed Computing*, 22:185-195, 2010.
- [2] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, ( 2d Edition), Wiley-Interscience, 414 pages, 2004.
- [3] Biely M., Robinson P. and Schmid U., Easy Impossibility Proofs for  $k$ -Set Agreement in Message-passing Systems. *Brief Announcement, Proc. 30th ACM Symposium on Principles of Distributed Computing (PODC'11)*, ACM Press, 2010.
- [4] Bonnet F. and Raynal M., A Simple Proof of the Necessity of the Failure Detector  $\Sigma$  to Implement an Atomic Register in Asynchronous Message-passing Systems. *Information Processing Letters*, 110(4):153-157, 2010.
- [5] Bonnet F. and Raynal M., On the Road to the Weakest Failure Detector for  $k$ -Set Agreement in Message-passing Systems. To appear in *Theoretical Computer Science*, 2011.
- [6] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for  $t$ -Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computation (STOC'93)*, San Diego (CA), pp. 91-100, 1993.
- [7] Bouzid Z. and Travers C., (Anti- $\Omega_k \times \Sigma_k$ )-Based  $k$ -Set Agreement Algorithms. *Proc. 12th Int'l Conference on Principles of Distributed Systems (OPODIS'10)*, Springer Verlag LNCS #6490, pp. 190-205, 2010.
- [8] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
- [9] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.
- [10] Chaudhuri S., More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105:132-158, 1993.
- [11] Delporte-Gallet C., Fauconnier H. and Guerraoui R., Tight Failure Detection Bounds on Atomic Object Implementations. *Journal of the ACM*, 57(4):Article 22, 2010.
- [12] Delporte-Gallet C., Fauconnier H., Guerraoui R. and Tielmann A., The Weakest Failure Detector for Message Passing Set-Agreement. *Proc. 22th Int'l Symposium on Distributed Computing (DISC'08)*, Springer-Verlag LNCS #5218, pp. 109-120, 2008.
- [13] Gafni E. and Kuznetsov P., The Weakest Failure Detector for Solving  $k$ -Set Agreement. *Proc. 28th ACM Symposium on Principles of Distributed Computing (PODC'09)*, ACM Press, pp. 83-91, 2009.
- [14] Guerraoui R. and Raynal M., The Alpha of Indulgent Consensus. *The Computer Journal*, 50(1):53-67, 2007.
- [15] Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.
- [16] Lo W.-K. and Hadzilacos V., Using Failure Detectors to Solve Consensus in Asynchronous Shared-memory Systems. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94, now DISC)*, Springer-Verlag LNCS #857, pp. 280-295, 1994.
- [17] Lynch N.A., *Distributed Algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, 1996.
- [18] Mostéfaoui A. and Raynal M.,  $k$ -Set Agreement with Limited Accuracy Failure Detectors. *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, ACM Press, pp. 143-152, 2000.
- [19] Mostéfaoui A. and Raynal M., Randomized Set Agreement. *Proc. 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*, ACM Press, pp. 291-297, 2001.
- [20] Mostéfaoui A., Raynal M. and Stainer J., Relations Linking Failure Detectors Associated with  $k$ -Set Agreement in Message-passing Systems. *Tech Report #1973*, 13 pages, IRISA, Université de Rennes (France), April 2011. Submitted to publication.
- [21] Neiger G., Failure Detectors and the Wait-free Hierarchy. *14th ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, Las Vegas -NV), 1995.
- [22] Raynal M., K-anti-Omega. *Rump Session at 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*, 2007.
- [23] Raynal M., Set agreement. *Encyclopedia of Algorithms*, Springer-Verlag, pp. 829-831, 2008 (ISBN 978-0- 387-30770-1).
- [24] Raynal M., Failure Detectors for Asynchronous Distributed Systems: an Introduction. *Wiley Encyclopedia of Computer Science and Engineering*, Vol. 2, pp. 1181-1191, 2009 (ISBN 978-0-471-38393-2).
- [25] Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. *Morgan & Claypool Publishers*, 251 pages, 2010 (ISBN 978-1-60845-293-4).
- [26] Raynal M., Fault-Tolerant Agreement in Synchronous Message-Passing Systems. *Morgan & Claypool Publishers*, 165 pages, 2010 (ISBN 978-1-60845-525-6).
- [27] Raynal M., Failure Detectors to solve Asynchronous  $k$ -set Agreement: a Glimpse of Recent Results. *The Bulletin of EATCS*, 103:75-95, 2011.
- [28] Raynal M. and Travers C., In Search of the Holy Grail: Looking for the Weakest Failure Detector for Wait-free Set Agreement. *Proc. 10th Int'l Conference On Principles Of Distributed Systems (OPODIS'06)*, Springer-Verlag LNCS #4305, pp. 1-17, 2006.
- [29] Saks M. and Zaharoglou F., Wait-Free  $k$ -Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.
- [30] Zielinski P., Anti-Omega: the Weakest Failure Detector for Set Agreement. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, pp. 55-64, Toronto (Canada), 2008.

## A Proof of Lemma 2

**Lemma 2** Let  $1 \leq y < n$ .  $\Pi\Sigma_{y+1,1} \not\leq_{\mathcal{AMP}} \bar{\Omega}_y$ .

**Proof** Supposing that there exists an algorithm  $A_y$  that builds  $\bar{\Omega}_y$  in  $\mathcal{AMP}[\Pi\Sigma_{y+1,1}]$  we show a contradiction, namely, there is an infinite run of  $A_y$  in which it is impossible to provide a valid output for  $\bar{\Omega}_y$ .

Let  $R$  be the set of the infinite runs of  $A_y$  such that :

- all the processes  $p_i$  with  $i > y + 1$  have initially crashed,
- the outputs of  $\Pi\Sigma_{y+1,1}$  at each  $p_i$ ,  $i \leq y + 1$ , are such that we always have  $qr_i = \{i\}$  and  $leader_i = i$ .

We show the following claim by induction on the number  $m$  of alive processes at time  $\tau$  in a run of  $R$ .

*Claim  $C(m)$ .* For each run  $r$  of  $R$  in which, at time  $\tau$ , the only processes alive are  $p_i$ ,  $1 \leq i \leq m$ , there is a run  $r' \in R$ , similar to  $r$  until  $\tau$ , in which  $\exists \tau' \geq \tau, \exists i_0 \in \{1, \dots, m\} : leaders_{i_0}^{\tau'} \supseteq \{1, \dots, m\}$ .

*Base case.* Let  $r$  be a run of  $R$  in which  $p_1$  is the only alive process at  $\tau$ . The weak eventual leadership property of  $\bar{\Omega}_y$  ensures that there is a time  $\tau'$  after which  $leaders_1 \ni 1$ , consequently,  $C(1)$  is verified.

*Induction step.* Assuming that  $C(m)$  is true for an  $m \geq 1$ , we show that it entails  $C(m+1)$ . Let  $r \in R$  be a run in which, at  $\tau$ , the only alive processes are  $p_i$ ,  $1 \leq i \leq m+1$ . Let  $r'$  be a run of  $R$  similar to  $r$  until  $\tau$  in which  $p_1$  crashes after  $\tau$ . According to the induction hypothesis (applied on renamed processes), there is a run  $r'_1$  of  $R$  similar to  $r'$  until  $\tau$  in which there is  $i_1 \in \{2, \dots, m+1\}$  and  $\tau_1 \geq \tau$  such that  $leaders_{i_1}^{\tau_1} \supseteq \{2, \dots, m+1\}$ .

Let now  $r_1$  be a run of  $R$ , similar to  $r$  until  $\tau$ , in which messages from  $p_1$  are delayed until  $\tau_1$ .  $r'_1$  and  $r_1$  are indistinguishable for processes  $p_i$ ,  $i \in \{2, \dots, m+1\}$ . Hence, in  $r_1$ , we also have  $leaders_{i_1}^{\tau_1} \supseteq \{2, \dots, m+1\}$ .

Repeating the same construction, we can now define a run  $r_2$  similar to  $r_1$  until  $\tau_1$ , in which there exist  $\tau_2 \geq \tau_1$  and  $i_2 \in \{1, \dots, m+1\} \setminus \{2\}$  such that  $leaders_{i_2}^{\tau_2} \supseteq \{1, \dots, m+1\} \setminus \{2\}$ .

We iterate the process, by delaying the messages from each process, one after the other. Thus, we obtain an infinite run of  $R$  in which, for all processes  $p_j$ ,  $j \in \{1, \dots, m+1\}$ , there is infinitely often a process  $p_i$  that verifies  $leaders_i \supseteq \{1, \dots, m+1\} \setminus \{j\}$ . Consequently, if in this run no process ever verifies  $leaders \supseteq \{1, \dots, m+1\}$ , then every process disappears infinitely often from a set  $leaders$ . That contradicts the weak eventual leadership property of  $\bar{\Omega}_y$  which ends the induction step.

*Contradiction.* By induction, the claim  $C(m)$  is true for all  $m \in \{1, \dots, y+1\}$ . But  $C(y+1)$  entails that, there is a run of  $R$  in which a process verifies  $leaders \supseteq \{1, \dots, y+1\}$ . That contradicts the validity property of  $\bar{\Omega}_y$  and proves the lemma.

□<sub>Lemma 2</sub>

Another look at that proof could be the following : (1)  $\Pi\Sigma_{y+1,1}$  provides no information on failures in the runs of  $R$ , (2)  $A_y$  simulates  $\bar{\Omega}_y$  in a wait-free manner among  $y+1$  processes in the runs of  $R$ . The contradiction then comes from the non-triviality of  $\bar{\Omega}_y$  in a system of  $y+1$  processes.

## B Implementation of $\text{Alpha}_x$ in $\mathcal{AMP}[\Sigma_x]$

This appendix presents an implementation of an  $\text{Alpha}_x$  object on top of  $\mathcal{AMP}[\Sigma_x]$ . This implementation is obtained from a modification of the algorithm proposed in [7] which is first described.

### B.1 The $\text{Alpha}_x$ object used by Bouzid and Travers [7]

**The obligation property used in [7]** The  $\text{Alpha}_x$  object used in [7] has the same specification as the one defined in Section 5.1 (which is close to the one defined in [14, 28]) but for the obligation property which (similarly to [14]) is defined as follows.

- **Obligation.** Let  $I = \text{propose}(r, -)$  be a terminating invocation. If every invocation  $I' = \text{propose}(r', -)$  that starts before  $I$  returns is such that  $r' < r$ , then  $I$  returns a non- $\perp$  value.

It is easy to see that this specification is not  $\Sigma_x$ -aware. In presence of concurrent invocations, it directs at most one process to decide a non- $\perp$  value, namely, the one with the highest round number. The current outputs of  $\Sigma_x$  are irrelevant in this statement.

**Principles: establish a priority on values** Each process  $p_i$  manages a local variable  $est_i$  (initialized to  $\perp$ ) that represents its current estimate  $v$  of the value it will decide and a pair  $(lre_i, pos_i)$  that defines the priority associated with  $v$  from  $p_i$ 's point of view (the aim is to decide values with the highest priority);  $lre_i = r$  means that  $r$  is the highest round seen by  $p_i$  and  $pos_i = \rho \in [1..2^r]$  is the position of  $v$  in round  $r$ . The pairs  $\langle r, \rho \rangle$  are used to establish a priority on proposed values. The function  $g(\rho, \delta) = 2^\delta(\rho - 1) + 1$  (where  $\delta$  is a difference between two round numbers) is used to compute the priority of a value in the following rounds. More precisely, let  $(r, \rho)$  and

$(r', \rho')$  (such that  $r \leq r'$ ) be the pairs associated with the values  $v$  and  $v'$ , respectively. Value  $v$  has lower priority than value  $v'$  at round  $r'$  iff  $g(\rho, r' - r) < \rho'$  or  $(g(\rho, r' - r) = \rho') \wedge (v < v')$ .

Our description of Bouzid-Travers's algorithm (algorithm 3) is schematic. The reader will refer to [7] for more detailed presentation and a proof. The implementation of the operation `propose()` is made up of two sequential phases: a read phase followed by write phase.

```

init  $lre_i \leftarrow 0$ ;  $est_i \leftarrow \perp$ ;  $pos_i \leftarrow 0$ .

operation propose( $r, v_i$ ):
(01) broadcast REQ_R( $r$ );
(02) repeat  $Q_i \leftarrow qr_i$ 
(03) until  $(\forall j \in Q_i : \text{RSP\_R}(r, \langle lre_j, pos_j, est_j \rangle) \text{ received from } p_j)$  end repeat;
(04) let  $rcv_i = \{ \langle lre_j, pos_j, est_j \rangle : \text{RSP\_R}(r, \langle lre_j, pos_j, est_j \rangle) \text{ received} \}$ ;
(05) if  $(\exists lre : \langle lre, -, - \rangle \in rcv_i : lre > lre_i)$  then return( $\perp$ ) end if;
(06)  $pos_i \leftarrow \max\{pos \mid \langle r, pos, v \rangle \in rcv_i\}$ ;  $est_i \leftarrow \max\{v \mid \langle r, pos_i, v \rangle \in rcv_i\}$ ;
(07) if  $(est_i = \perp)$  then  $est_i \leftarrow v_i$  end if;
(08) while( $pos_i < 2^r$ ) do
(09)    $pos_i \leftarrow pos_i + 1$ ;  $pst_i \leftarrow pos_i$ ; % this line is executed atomically %
(10)   broadcast REQ_W( $r, pst_i, est_i$ );
(11)   repeat  $Q_i \leftarrow qr_i$ 
(12)   until  $(\forall j \in Q_i : \text{RSP\_W}(r, pst_i, \langle lre_j, pos_j, est_j \rangle) \text{ received from } p_j)$  end repeat;
(13)   let  $rcv_i = \{ \langle lre_j, pos_j, est_j \rangle : \text{RSP\_W}(r, pst_i, \langle lre_j, pos_j, est_j \rangle) \text{ received} \}$ ;
(14)   if  $(\exists lre : \langle lre, -, - \rangle \in rcv_i : lre > r)$  then return( $\perp$ ) end if;
(15)    $pos_i \leftarrow \max\{pos \mid \langle r, pos, v \rangle \in rcv_i\}$ ;  $est_i \leftarrow \max\{v \mid \langle r, pos_i, v \rangle \in rcv_i\}$ 
(16) end while;
(17) return( $est_i$ ).

when REQ_R( $rd$ ) received from  $p_j$ :
(18) if  $rd > lre_i$  then  $pos_i \leftarrow g(pos_i, rd - lre_i)$ ;  $lre_i \leftarrow rd$  end if;
(19) send RSP_R( $rd, \langle lre_i, pos_i, est_i \rangle$ ) to  $p_j$ .

when REQ_W( $rd, pos, est$ ) received from  $p_j$ :
(20) if  $rd \geq lre_i$  then  $pos_i \leftarrow g(pos_i, rd - lre_i)$ ;  $lre_i \leftarrow rd$ 
(21)   case  $pos_j > pos_i$  then  $est_i \leftarrow est$ ;  $pos_i \leftarrow pos$ 
(22)    $pos_j = pos_i$  then  $est_i \leftarrow \max\{v_i, est\}$ 
(23)    $pos_j < pos_i$  then nop
(24)   end case
(25) end if;
(26) send RSP_W( $rd, pos, \langle lre_i, pos_i, est_i \rangle$ ) to  $p_j$ .

```

Algorithm 3:  $\text{Alpha}_k$  in  $\mathcal{AMP}[\Sigma_k]$ : Bouzid-Travers's implementation [7]

**Succinct description of the algorithm: the read phase** When it invokes `propose( $r, v$ )`, a process  $p_i$  first broadcasts a read-request message (line 01) to (a) obtain information on values proposed in previous rounds (if any) and (b) learn if other processes have started higher rounds.

When a process  $p_j$  receives such a message `REQ_R( $rd$ )` (where  $rd$  is a round number) it redefines its pair  $\langle lre_j, pos_j \rangle$  if  $rd > lre_j$  (line 18) (the new position  $pos_j$  of  $est_j$  is re-computed according to the values of  $rd$  and  $lre_j$ ). In all cases,  $p_j$  sends back an answer to  $p_i$  carrying its current value  $est_j$  and the associated pair  $\langle lre_j, pos_j \rangle$  (line 19).

Then, when it has received a response from each process in its current quorum  $qr_i$  as supplied by  $\Sigma_x$  (lines 02-03),  $p_i$  returns  $\perp$  if it has received an answer indicating that another process has started a round higher than  $lre_i$  (lines 04-05). Otherwise,  $lre_i = r$  is the greatest round number known by  $p_i$ . In that case,  $p_i$  update  $pos_i$  to the greatest position associated with round  $r$  it has seen and adopts the corresponding value  $v$  as its current estimate  $est_i$  (line 06). Moreover, if  $v = \perp$ ,  $p_i$  adopts  $v_i$  into  $est_i$ , namely, the value it proposes to the  $\text{Alpha}_x$  object (line 07).

**Succinct description of the algorithm: the write phase** Process  $p_i$  enters then a loop that it will exit either by returning  $\perp$  (line 14) or its current estimate value (line 17). The maximum number of times that this loop can be executed depends on the round number  $r$  and the current position value  $pos_i$  (line 08). The part of the loop body defined by lines 10-15 is the same as lines 01-06. The difference is that, instead of obtaining information on the current state,  $p_i$  cooperate with the processes of its current quorum  $qr_i$  in order to try to increase the priority of its current  $est_i$ . Hence instead of a read-request,  $p_i$  broadcasts write-request messages.

Each time a process  $p_j$  receives such a message that carries a triplet  $\langle rd, pos, val \rangle$  it updates its current state in order this local state contains the value with the highest priority (and the associated control data). Operationally, if  $rd \geq lre_j$ ,  $p_j$  first updates  $pos_j$  and  $lre_j$  (line 20) exactly as it did at line 18 when it received a read-request message. Then, according to the value of  $pos_j$  and  $pos$  (lines 21-24),  $p_j$  updates  $est_j$  and  $pos_j$  if  $pos > pos_j$  or updates only  $est_j$  if  $pos_j = pos$ . In all cases,  $p_j$  sends back a response carrying its local state to the process that sent the write-request.

As already indicated, a proof that this algorithm implements an  $\text{Alpha}_x$  object that satisfies the round-based obligation property stated at the beginning of this section is given in [7].

## B.2 An implementation of $\text{Alpha}_x$ as defined in Section 5.1

Algorithm 4 describes an implementation of the  $\Sigma_x$ -aware specification of  $\text{Alpha}_x$  defined in Section 5.1. This algorithm is an appropriate improvement of Algorithm 3.

To make the presentation easier, the line numbers are the same in both algorithms. The lines that are new or modified are prefixed by the letter N. These modifications concern message filtering. This filtering is used to prevent a process  $p_i$  from sending read or write-requests to the processes  $p_j$  such that  $p_i$  does not need information from  $p_j$  to complete its current invocation of  $\text{propose}()$ . Hence, such a process  $p_j$  cannot direct  $p_i$  to return  $\perp$  while it could return a non- $\perp$  value (and additionally the values not sent by  $p_i$  cannot force other processes to return  $\perp$ ).

**Modification of the read phase** A process  $p_i$  records in  $r\_req\_set_i$  the set of processes to which it has already sent a  $\text{REQ\_R}(r)$  message (lines N01 and N02-2) and sends read-request messages  $\text{REQ\_R}(r)$  only to the processes  $p_j$  that belong to its quorum  $qr_i$  (line N02-1).

Then, when it stops waiting for response messages, it considers only the responses sent by the processes of its last quorum plus its own response (line N04).

**Modification of the write phase** The message exchange pattern of that phase is modified similarly to what has been done for the read phase (lines N10, N12-1, N12-2 and N13).

```

init  $lre_i \leftarrow 0$ ;  $est_i \leftarrow \perp$ ;  $pos_i \leftarrow 0$ .

operation propose( $r, v_i$ ):
(N01)   $r\_req\_set_i \leftarrow \emptyset$ ;
(02)  repeat  $Q_i \leftarrow qr_i$ ;
(N02-1) for each  $j \in Q_i \setminus r\_req\_set_i$  do send  $\text{REQ\_R}(r)$  to  $p_j$  end for;
(N02-2)  $r\_req\_set_i \leftarrow r\_req\_set_i \cup Q_i$ 
(03)  until ( $\forall j \in Q_i : \text{RSP\_R}(r, \langle lre_j, pos_j, val_j \rangle)$  received from  $p_j$ ) end repeat;
(N04)  let  $rcv_i = \{ \langle lre_j, pos_j, est_j \rangle : \text{RSP\_R}(r, \langle lre_j, pos_j, est_j \rangle) \text{ received from } Q_i \}$ ;
(05)  if ( $\exists lre : \langle lre, -, - \rangle \in rcv_i : lre > lre_i$ ) then return( $\perp$ ) end if;
(06)   $pos_i \leftarrow \max\{pos \mid \langle r, pos, v \rangle \in rcv_i\}$ ;  $est_i \leftarrow \max\{v \mid \langle r, pos_i, v \rangle \in rcv_i\}$ ;
(07)  if ( $est_i = \perp$ ) then  $est_i \leftarrow v_i$  end if;
(08)  while ( $pos_i < 2^r$ ) do
(09)     $pos_i \leftarrow pos_i + 1$ ;  $pst_i \leftarrow pos_i$ ; % this line is executed atomically %
(N10)     $w\_req\_set_i \leftarrow \emptyset$ ;
(11)    repeat  $Q_i \leftarrow qr_i$ ;
(N12-1) for each  $j \in Q_i \setminus w\_req\_set_i$  do send  $\text{REQ\_W}(r, pst_i, est_i)$  to  $p_j$  end for;
(N12-2)  $w\_req\_set_i \leftarrow w\_req\_set_i \cup Q_i$ 
(12)    until ( $\forall j \in Q_i : \text{RSP\_W}(r, pst_i, \langle lre_j, pos_j, val_j \rangle)$  received from  $p_j$ ) end repeat;
(N13)     $rcv_i \leftarrow \{ \langle lre_j, pos_j, est_j \rangle : \text{RSP\_W}(r, pst_i, \langle lre_j, pos_j, est_j \rangle) \text{ received from } Q_i \}$ ;
(14)    if ( $\exists lre : \langle lre, -, - \rangle \in rcv_i : lre > r$ ) then return( $\perp$ ) end if;
(15)     $pos_i \leftarrow \max\{pos \mid \langle r, pos, v \rangle \in rcv_i\}$ ;  $est_i \leftarrow \max\{v \mid \langle r, pos_i, v \rangle \in rcv_i\}$ 
(16)  end while;
(17)  return( $est_i$ ).

when  $\text{REQ\_R}(rd)$  received from  $p_j$ :
(18)  if  $rd > lre_i$  then  $pos_i \leftarrow g(pos_i, rd - lre_i)$ ;  $lre_i \leftarrow rd$  end if;
(19)  send  $\text{RSP\_R}(rd, \langle lre_i, pos_i, est_i \rangle)$  to  $p_j$ .

when  $\text{REQ\_W}(rd, pos, est)$  received from  $p_j$ :
(20)  if  $rd \geq lre_i$  then  $pos_i \leftarrow g(pos_i, rd - lre_i)$ ;  $lre_i \leftarrow rd$ 
(21)    case  $pos_j > pos_i$  then  $est_i \leftarrow est$ ;  $pos_i \leftarrow pos$ 
(22)       $pos_j = pos_i$  then  $est_i \leftarrow \max\{v_i, est\}$ 
(23)       $pos_j < pos_i$  then nop
(24)    end case
(25)  end if;
(26)  send  $\text{RSP\_W}(rd, pos, \langle lre_i, pos_i, est_i \rangle)$  to  $p_j$ .

```

Algorithm 4:  $\text{Alpha}_k$  in  $\mathcal{AMP}[\Sigma_k]$  as defined in Section 5.1

**Theorem 6.** Algorithm 4 implements an  $\text{Alpha}_x$  object as defined in Section 5.1.

**Proof** It is easy to see that the lines that are new or modified do not add spurious values, hence the proof of the validity property is the same as the one given in [7].

The same holds for the quasi-agreement property. This follows from the observation that, as the channels are asynchronous, the messages that are sent in Algorithm 3 and not sent in Algorithm 4 can be received too late in Algorithm 3, namely, after the processes have invoked the `return()` statement and decided. The quasi-agreement property follows from this observation and the proof given in [7].

As far as the termination property is concerned, let us first observe that, due to line 09, a process will exit the **while** loop (lines 08-16) if it does not loop forever in the **repeat** loop of lines 11-12. Hence, the proof of the termination property consists in showing that no correct process loops forever in the **repeat** loop of lines 02-03 or the **repeat** loop of lines 11-12.

Let us first consider the **repeat** loop of lines 02-03 and assume that a correct process  $p_i$  loops forever. It follows from the liveness property of  $\Sigma_x$  that there is a time  $\tau$  after which  $qr_i$  contains only correct processes. Hence, due to lines 02-03 and the fact that the number of processes is finite, it follows that there is a time  $\tau' \geq \tau$  after which  $p_i$  has sent a read-request message `REQ_R(r)` to each correct process that belongs to  $qr_i$  (whatever the value of  $qr_i$ ). As each correct process sends by return a matching response message `RESP_R(r, -)`, it follows that  $p_i$  eventually receives from each process in  $Q_i$  a message `RESP_R(r, -)` matching its `REQ_R(r)` request, which completes the proof that the **repeat** loop of the read phase always terminates.

For the **repeat** loop of lines 11-12, let us first observe that the local variable  $pst_i$  (which is initialized to the current value of  $pos_i$  before entering the loop, line 09) is not modified during the execution of that **repeat** loop. Observing that  $p_i$  repeatedly sends then a write-request message `WRITE_W(r, pst_i, -)` and wait for matching response messages `RESP_W(r, pst_i, -)`, the same reasoning as previously applies from which we conclude that  $p_i$  eventually exits the **repeat** loop of the write phase.

For the obligation property, let us remember the following definition:  $p_\ell$  being a correct process  $Q(\ell, \tau)$  denotes the set  $\{i \in \mathcal{C} \mid \forall \tau_i, \tau_\ell \geq \tau : qr_i^{\tau_i} \cap qr_\ell^{\tau_\ell} = \emptyset\}$ . We have to show that if, after some time  $\tau$ , (a) only  $p_\ell$  and processes of  $Q(\ell, \tau)$  invoke `propose()` and (b)  $p_\ell$  invokes `propose()` infinitely often, then at least one invocation issued by  $p_\ell$  returns a non- $\perp$  value.

Let  $\Pi(\ell, \tau) = \cup_{\tau' \geq \tau} qr_\ell^{\tau'}$  and  $\bar{\Pi}(\ell, \tau) = \cup_{\tau' \geq \tau, i \in Q(\ell, \tau)} qr_i^{\tau'}$ . It follows from the definitions of the  $qr_i$  sets and  $Q(\ell, \tau)$  that  $\Pi(\ell, \tau) \cap \bar{\Pi}(\ell, \tau) = \emptyset$ .

Let  $\tau_0 > \tau$  be a time instant such that each invocation of the operation `propose()` issued before  $\tau$  has returned or crashed and all the request and response messages generated by these invocations have been received and processed.

Let us observe that in Algorithm 4, a process sends requests and receives responses only to or from processes in its quorum. Consequently, after time  $\tau_0$ , (a) process  $p_\ell$  sends requests and receives responses only to or from processes in  $\Pi(\ell, \tau)$ ; and (b) processes in  $Q(\ell, \tau)$  sends requests and receives responses only to or from processes in  $\bar{\Pi}(\ell, \tau)$ . Moreover, as  $\Pi(\ell, \tau) \cap \bar{\Pi}(\ell, \tau) = \emptyset$ , for the processes in  $\Pi(\ell, \tau)$  such a run  $R$  cannot be distinguished from a run  $R'$  in which the processes in  $\bar{\Pi}(\ell, \tau)$  have crashed by time  $\tau_0$  (Observation OB).

Since after  $\tau_0$  (a)  $p_\ell$  invokes infinitely often `propose()`, (b) the processes in  $\Pi(\ell, \tau)$  do not invoke `propose()`, and (c) the invocations `propose()` issued by  $p_\ell$  are done with strictly increasing round numbers, it follows that one of these invocations  $I = \text{propose}()$  carries a round number greater than all those seen before  $\tau$  by the processes in  $\Pi(\ell, \tau)$ .

Let us consider the run in which the processes whose identities belong to  $\bar{\Pi}(\ell, \tau)$  have crashed by  $\tau_0$ . When, in that run,  $p_\ell$  issues invocation  $I$ , the round-based obligation property used in [7] is satisfied and  $p_\ell$  returns a non- $\perp$  value (Lemma 5 in [7]). Due to Observation OB on the indistinguishability between  $R$  and  $R'$  for process  $p_\ell$ , it follows that  $p_\ell$  returns the same non- $\perp$  value in run  $R$ , which concludes the proof of the obligation property.  $\square_{\text{Theorem 6}}$

## C Proof of Theorem 4

**Theorem 4** Algorithm 2 solves the  $x$ -set agreement problem in  $\mathcal{AMP}[\Pi\Sigma_{x,1}]$ .

**Proof** Validity and agreement properties. Let us first observe that, due to the test of line 02, the default value  $\perp$  cannot be decided. The fact that a decided value is a proposed value follows then from the validity of the underlying  $\text{Alpha}_x$  object. Similarly, the fact that at most  $k$  non- $\perp$  values are decided follows directly from the quasi-agreement property of the underlying  $\text{Alpha}_x$  object.

Termination property. It follows from the reliable broadcast operation that, at soon as a process decides (invokes `return()`) each correct process eventually delivers the same `DECISION(d)` message and decides (if not yet done). The proof is by contradiction: assuming that no process decides, we show that at least one correct process executes `rel_broadcast()` (and consequently, all correct processes decide).

Let  $p_\ell$  be a correct process that appears in the definition of the eventual partial leadership property of  $\Pi\Sigma_x$ . It follows from the definition of  $p_\ell$  that we eventually have forever  $leader_\ell = \ell$ .

Let  $R_\ell$  be the set of the identities of the processes  $p_j$  (with  $j \neq \ell$ ) such that we have  $leader_j = j$  infinitely often. It follows from the contrapositive of the eventual partial leadership property of  $\Pi\Sigma_x$  that there is a time  $\tau_{R_\ell}$  such that  $\forall j \in R_\ell, \forall \tau_1, \tau_2 \geq \tau_{R_\ell} : qr_j^{\tau_1} \cap qr_\ell^{\tau_2} = \emptyset$ , from which we conclude that  $R_\ell \subseteq Q(\ell, \tau_{R_\ell})$  (this is the set defined in the obligation property of  $\text{Alpha}_x$ ).

Let us notice that, due to test of line 03, there is a finite time  $\tau_a$  after which the only processes that invoke  $\text{Alpha}_x.\text{propose}()$  are the processes in  $R_\ell \cup \{\ell\}$ . Moreover (as by the contradiction assumption no process decides) it follows that, after  $\tau_a$ ,  $p_\ell$  invokes  $\text{Alpha}_x.\text{propose}()$  infinitely often. Let  $\tau_b$  be a time greater than  $\max(\tau_{R_\ell}, \tau_a)$  from which we have  $R_\ell \subseteq Q(\ell, \tau_{R_\ell}) \subseteq Q(\ell, \tau_b)$ .

As after  $\tau_b$  (a) only processes in  $R_\ell \cup \{\ell\}$  invoke  $\text{Alpha}_x.\text{propose}()$ , (b)  $p_\ell$  invokes  $\text{Alpha}_x.\text{propose}()$  infinitely often and (c)  $R_\ell \subseteq Q(\ell, \tau_b)$ , we conclude from the obligation property of  $\text{Alpha}_x$  that at least one invocation of  $p_\ell$  returns a value  $d \neq \perp$  and consequently executes  $\text{rel\_broadcast\_DECISION}(d)$ . This contradicts the fact that no process decides and concludes the proof of the theorem.  $\square_{\text{Theorem 4}}$